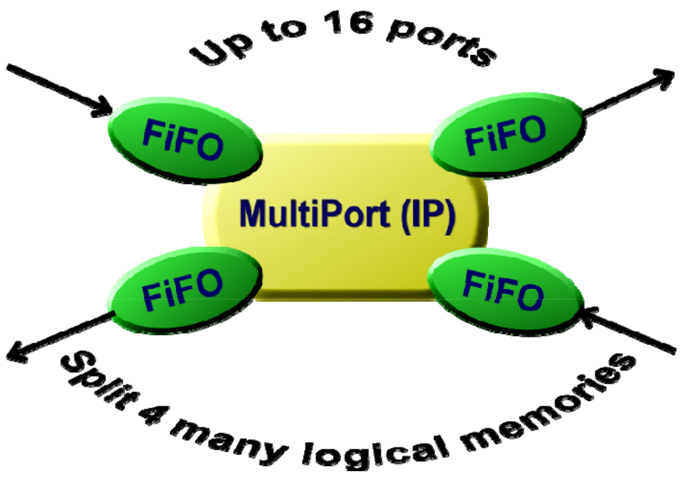
GIDEL

PROCMultiPort IP

SDRAM Controller



User's Guide

October 2010

USA
1600 Wyatt Drive Suite 1
Santa Clara,
CA 95054, USA
Tel: 1 - 408 - 969 - 0389
Fax: 1 - 866 - 615 - 6810

sales_usa@GiDEL.com

Worldwide
2 Ha'ilan Street, P.O. Box 281
Or Akiva,
Israel 30600
Tel: +972 - 4610 - 2500
Fax: +972 - 4610 - 2501

sales_eu@GiDEL.com

Web: www.GiDEL.com
info@GiDEL.com

Contents

Figures

Tables

Introduction

GiDEL PROCMultiPort™ is an advanced DRAM memory controller that enables easy and efficient usage of the memory featured on GiDEL PROC boards. PROCMultiPort was designed to utilize the on-board memory in the most efficient way while keeping the interface simple. With ***PROCMultiPort***, a whole inventory of special-purpose memories may be replaced with standard on-board DRAM memory blocks and FPGA internal memory, while keeping a common interface. This eliminates the possibility of a device becoming obsolete because it uses a certain type of memory. PROCMultiPort provides:

- ✓ A simple and convenient way to achieve high system performance with the on-board SDRAM / DDR / DDR II memories.
- ✓ Reduction of costs and of the PCB area, by using a single SDRAM / DDR / DDR II device in conjunction with the PROCMultiPort core, instead of adding multiple external memories.

PROCMultiPort converts the on-board memory to a true multi-port memory. A single memory block may be accessed simultaneously by up to 16 Ports, each Port working in its own clock domain and with its own data width. Sequential access mode enables accessing the memory in infinite bursts (regardless the memory type) or word-by-word, using convenient FIFO-like interface. Moreover, using ***PROCMultiPort*** with the on-board DMA channels enables "fire-and-forget" approach. Now, the user may initiate fast data transfers and free the host CPU for other tasks until the transfer is over.

A number of advanced features of ***PROCMultiPort***, such as Segmented Data Access and Port Status Flags, enable pushing the memory performance to the limits. By dividing data into two-dimensional arrays, ***PROCMultiPort*** segmented mode logically enlarges the FPGA memory size. An additional feature called Memory Bandwidth Controller allows the user to fully utilize the total memory bandwidth on a PROC board. With this feature, the MultiPort ports automatically adjust their data transfer rate in order to best fit the available memory bandwidth.

GiDEL PROCMultiPort may open way for new technologies. With ***PROCMultiPort***, one on-board memory block may replace several external memory blocks, while providing high throughput. Complex designs may be simplified or completely replaced by ***PROCMultiPort***. For example, ***PROCMultiPort*** can replace swappable double buffers or implement multiple logical memories in the same physical memory. Among the applications that might utilize this memory controller are: simulator data storages, huge FIFOs, data processing engines, 3D applications, long delay lines, diagnostic buffers and more.

PROCMultiPort may enable acquisition systems, display systems etc. to maintain constant operation controlled by software without the need for the software to respond in real-time.

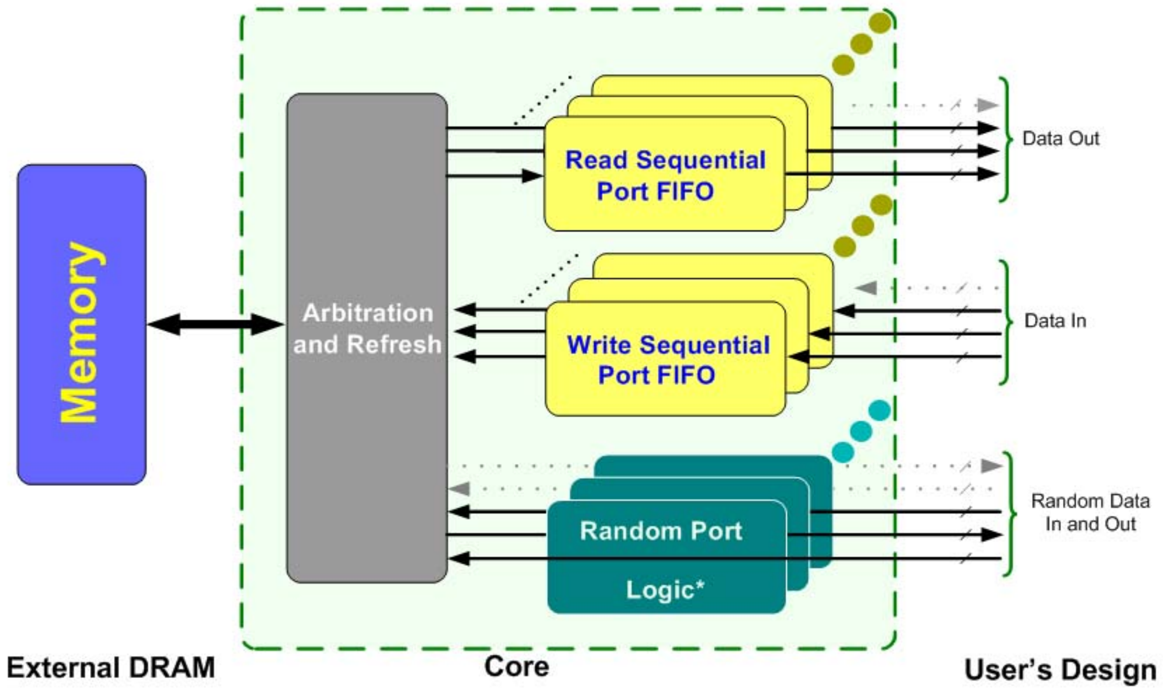
Added to this, the data transferred over the MultiPort channels may now be logically divided into fields. This way, the data read from the MultiPort may be saved into a file and then loaded by another application.

PROCMultiPort IP Key Features

- ✓ Up to 16 ports connected to one memory block.
- ✓ All the ports (or part of them) may operate continuously and simultaneously.
- ✓ Each port (and each memory block) may work in its own clock domain.
- ✓ Each port (and each memory block) can work with a different data width.
- ✓ Each port may be forced to access a different memory area and limited to a certain address space, thus logically splitting the memory block into several memories.
- ✓ Simple interface (similar to synchronous FIFO) for sequential ports.
- ✓ Straightforward random memory access with random ports.
- ✓ Advanced Segmented working mode option for sequential ports.
- ✓ Memories zero wait state.
- ✓ Allows burst DRAM accesses with an unlimited burst length.
- ✓ Automatic Memory Bandwidth Controller
- ✓ Automatic arbitration between ports.
- ✓ Automatic generation of refresh cycles in the background.
- ✓ Automatic generation of ***PROCMultiPort*** instances and their integration into user's design using ***GiDEL PROCWizard*™.**

✓ Using GIDEL ***PROCWizard***, the MultiPort data may be split into logical fields and saved to file.

This section schematically shows the signal flow when using ***PROCMultiPort*** IP. Each sequential Port has an internal FIFO and each Random Port has internal logic that enables transparent access to the memory. The ***PROCMultiPort*** arbitrates between the Ports, granting Random Ports the highest priority. This enables working with all the Ports simultaneously, while allowing each Port to use in its own clock domain. An additional mechanism enables using different data widths for each Port. This mechanism automatically splits / concatenates user's data to fit the physical memory data width.



* Only one Random Port may be added automatically using GiDEL PROCWizard.

Figure 1 – PROCMultiPort Block Diagram

Note

Random accesses cause major degradation of the overall throughput. Therefore, it is advised to make all the ports sequential when possible. Random ports are mainly used for Flash emulation.

The core of PROCMultiPort controller has two basic interfaces:

- **Memory interface,** directly connected to the SDRAM memory. ***PROCWizard*** creates this interface automatically.
- **Port interface** (**user interface**)**,** connected to the software or to user's logic. Each Port has its own user interface. This includes data width, clock domain, access type and direction and more.

Port interfaces are configured by parameters. These parameters are defined using ***GiDEL PROCWizard***. After all the parameters are set, ***PROCWizard*** automatically generates a wrapper with individual interfaces for each Port, including all the I/Os that are required for the current project.

To set up the ***PROCMultiPort*** IP in ***GiDEL PROCWizard***, first enter the Configuration Screen. The Configuration Screen is used to actually build the design. New items can be added to the design in this mode and existing items can be redesigned. To enter the Configuration mode, click the button in ***PROCWizard*** toolbar, as shown in the figure below:

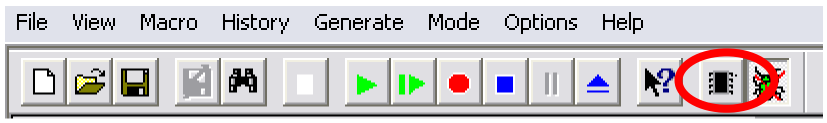


Figure 2 – Configuration mode icon

Once in Configuration Mode, you will see the design you are building represented by an item tree.

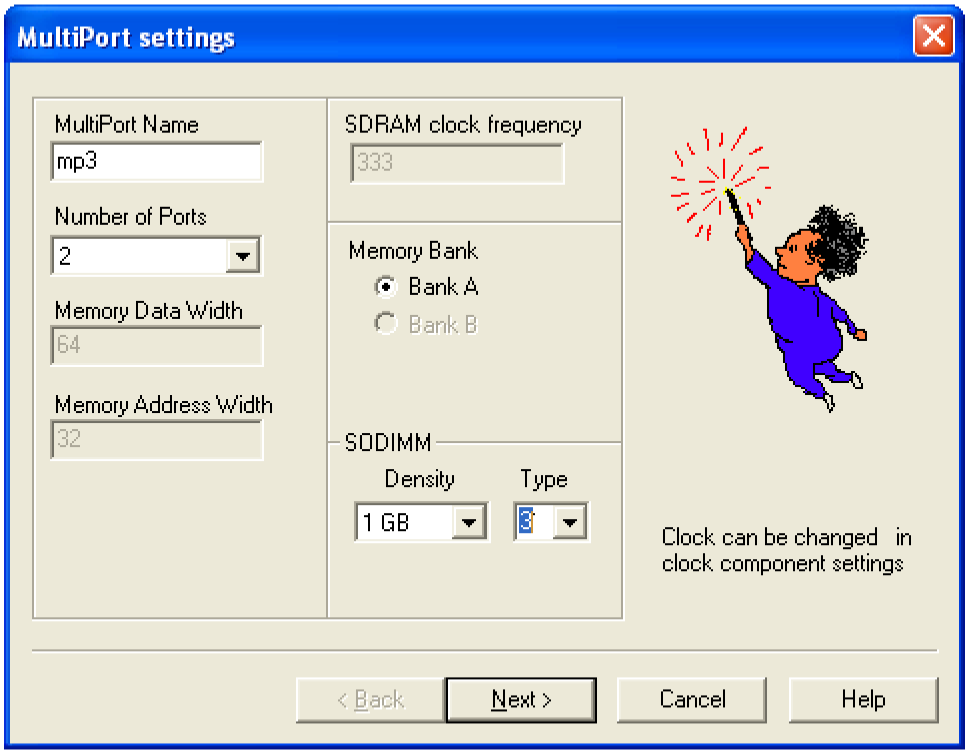
To learn more about these items, please refer to ***PROCWizard*** **User Manual**.

To add ***PROCMultiPort IP*** to an existing design, choose the IC item you wish to add this IP to.

Note

The FPGA device represented by this IC must be connected to the memory block you wish to use.

Point to the IC, right click, and choose from the pop-up menu **IP Core → Multiport**. This will start the ***CoreWizard*** and open the window shown in Figure 3. The ***CoreWizard*** will guide you through the process of customizing the ***PROCMultiPort*** IP core to suit the application.



MultiPort settings

MultiPort Name: mp3

Number of Ports: 2

Memory Data Width: 64

Memory Address Width: 32

SDRAM clock frequency: 333

Memory Bank: (● Bank A) ○ Bank B

SODIMM

Density: 1 GB

Type: 3

Clock can be changed in clock component settings

Figure 3 – MultiPort Settings Dialog

The **MultiPort setting** dialog box allows you to define the **MultiPort Name**, **Number of Ports** to be used, **SDRAM clock frequency**, **Memory Data Width** and the **Memory Bank** to be used.

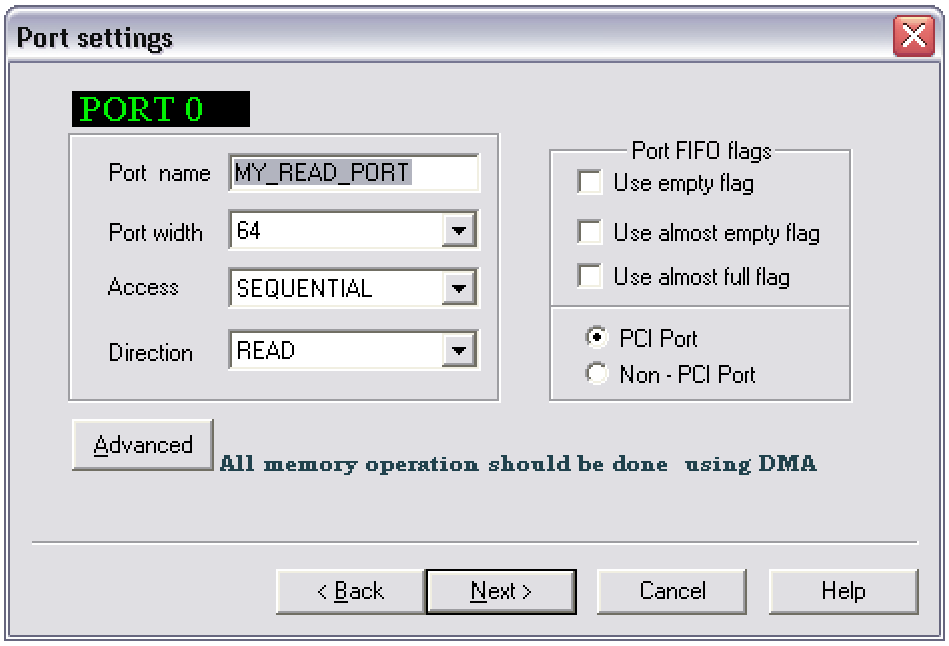
If the selected **Memory Bank** is a SODIMM memory, then a **SODIMM** group box with the fields **Density** and **Type** will appear in the MultiPort dialog box. (See Figure 3).

- Click the **Density** arrow to select one of the possible memory densities.
- Click the SODIMM **Type** arrow to select your SODIMM's memory type. To determine your SODIMM's type, refer to the ***GiDEL SODIMM Type Datasheet*** document. In accordance to the SODIMM's **Type,** the PROCWizard will automatically generate a customized design.

Click **Next** to open the **Port settings** dialog box.

Defining Basic Port Settings

In the **PORT settings** dialog box, you define the basic parameters of the current Port Clicking the **Next** and **Back** buttons enables to move between ports.


Port settings

PORT 0

Port name: MY_READ_PORT
Port width: 64
Access: SEQUENTIAL
Direction: READ

Port FIFO flags
- [ ] Use empty flag
- [ ] Use almost empty flag
- [ ] Use almost full flag

(●) PCI Port
( ) Non - PCI Port

All memory operation should be done using DMA

Figure 4 - Port Settings Dialog

The Port basic parameters include name, data width, access type (random or sequential) and direction (read or write).

Note

64bits or 128 bits-wideSequential Ports, in accordance to the board type, always use DMA to transfer data. 32bits-wide Sequential Ports always use direct access to transfer data.

In addition, you may define whether to use the **Port status flags** in the design. The Port status flags behave as follows:

- The **empty** flag goes high when the Port FIFO becomes empty.
- The **almost empty** flag goes high when the Port FIFO becomes less than 1/8 full.
- The **almost full** flag goes high when the Port FIFO becomes more than 7/8 full.

Finally, specify the Port source / destination (PCI/E or user logic). When **PCI/E Port** is chosen, the ***PROCWizard*** automatically connects the Port interface to the local bus logic, thus enabling accessing the Port directly from the host PC.

Note

Random Ports are non-PCI/E Ports by definition

Selecting **Non-PCI/E Port** opens the following window:

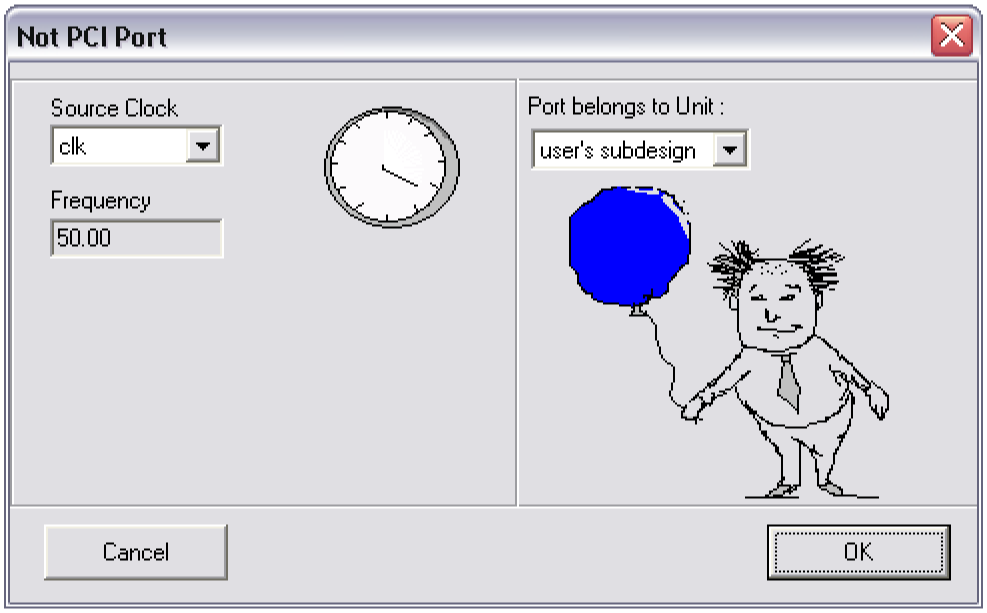


Figure 5 – Non-PCI/E Port Settings Dialog

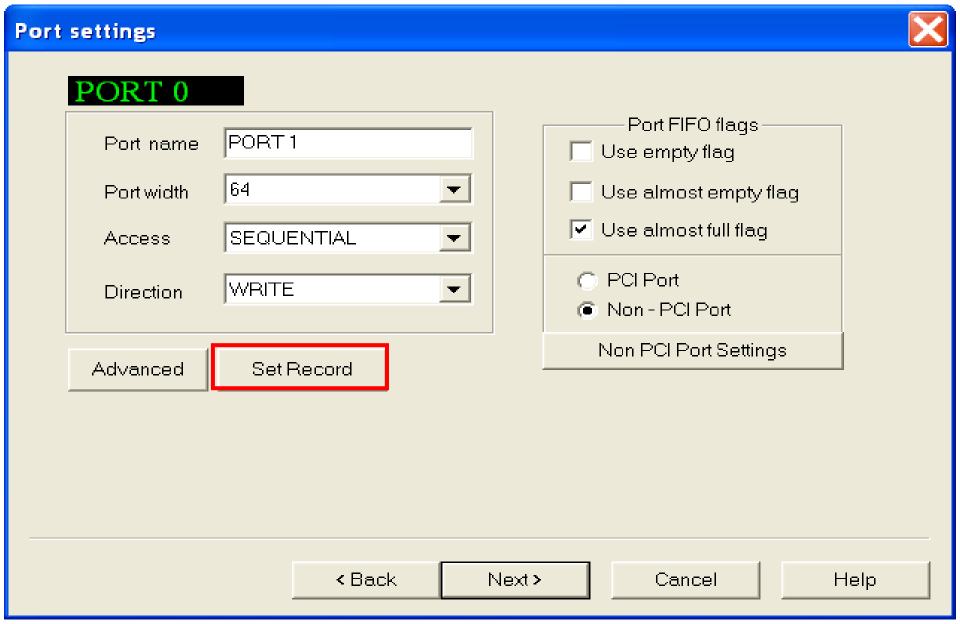
In the **Port belongs to Unit** box, select the destination subdesign (Entity / Module). Click the **Source Clock** arrow to select a port clock. If you wish to use your own clock, rather than a global clock, you must specify the clock's frequency in the **Frequency** box.

Note

To define a non-PCI/E Port, you must create a subdesign (module / entity) first. This subdesign must be connected to at least one clock that will be used to synchronize the Port with the subdesign

Click OK to return to the **Port settings** dialog box.

For a non-PCI/E Port, if the **Direction** is set to **Write**, in the **Port settings** dialog box will appear an additional button, the **Set Record** button, enabling to create a record and define fields.



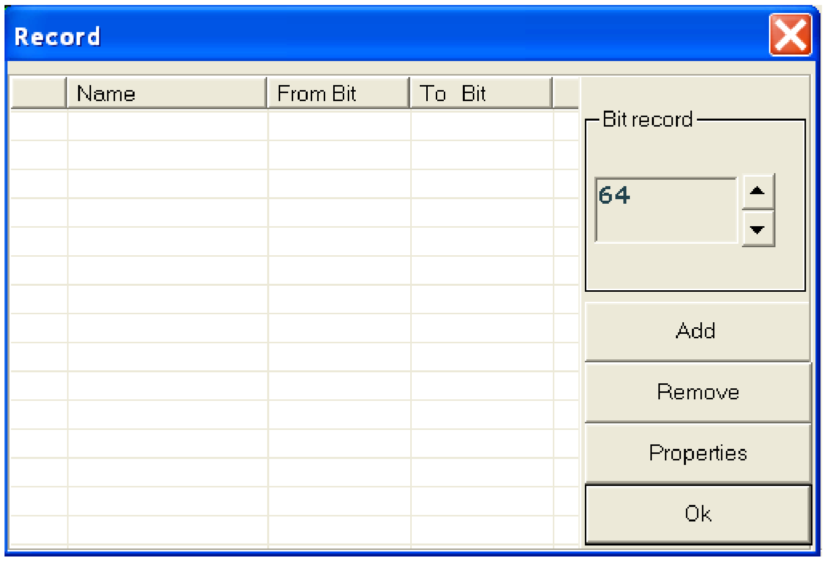
Port name: PORT 1
Port width: 64
Access: SEQUENTIAL
Direction: WRITE

☐ Use empty flag
☐ Use almost empty flag
☑ Use almost full flag

PCI Port [✓ Non - PCI Port]

Figure 6: The Set Record Button

Clicking on the **Set Record** button, opens the following dialog box:



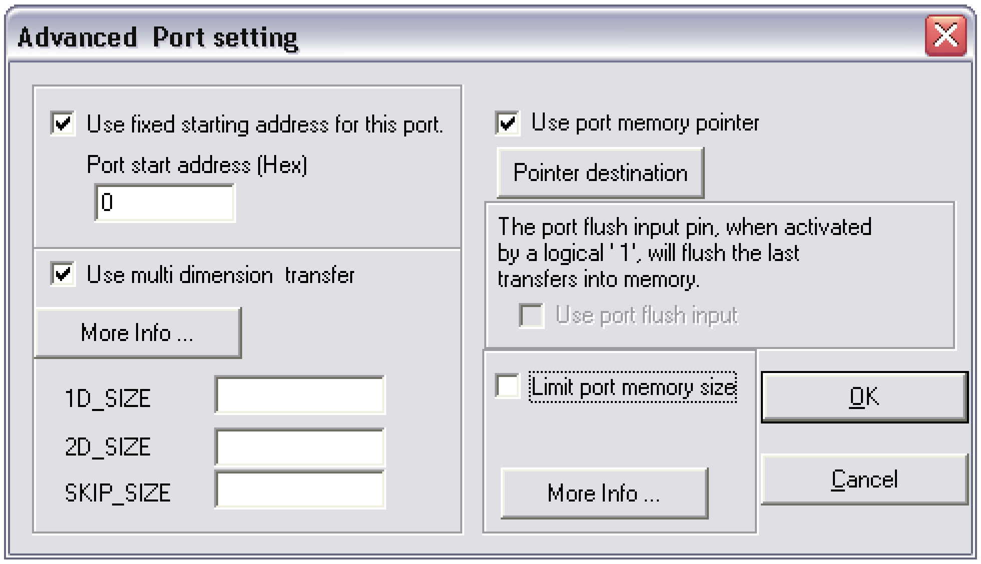
Bit record: 64

Figure 7: Record Dialog box for Non-PCI/E Write

Select the **Bit record** size by clicking on the up and down arrows and then click **Add** to define a record field. In the **Field** dialog box, define the field's name and its respective bits. Once a field is defined, it will appear in the **Record** dialog box's list. To delete a field, click on the field and then click **Remove**. To change a field's properties, either double click on the field or select and click the **Properties** button and **Field** dialog box will open. The record and its respective fields will be automatically incorporated into the project's h. file, accessible to the user's application.

Defining Advanced Port Settings

Advanced Port settings can be displayed by clicking the **Advanced** button in the Basic Port Settings dialog box. The Advanced Port Setting dialog box enables to define the starting address of the Port, limit the memory size to be used and set the parameters associated with the multi-dimensional transfer function.



Advanced Port setting

[✓] Use fixed starting address for this port.

Port start address (Hex): 0

[✓] Use multi dimension transfer

1D_SIZE

2D_SIZE

SKIP_SIZE

[✓] Use port memory pointer

The port flush input pin, when activated by a logical ' 1', will flush the last transfers into memory.

[ ] Use port flush input

[ ] Limit port memory size

Figure 8 - Advanced Port Settings Dialog

Selecting **Use fixed starting address** forces the Port to return to the specified address upon each Port reset (**Start** signal asserted high). If this option is not chosen, user's logic must set the Port address upon each reset. An additional bus will be added to the top-level design for this purpose.

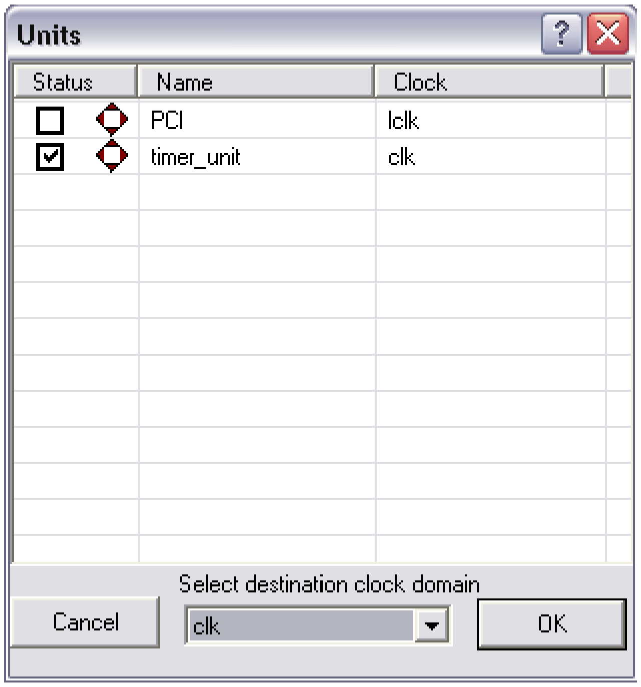
Selecting **Use multi dimension transfer** sets the Port to Segmented mode. This mode enables to perform 2D data transfers. In this mode, the Port accesses several words (a row) sequentially and then skips several words. This is repeated for a number of rows and then the Port asserts the **ready** signal high.

- The row size is set by **1D_SIZE** parameter.
- The number of rows to be accessed is set by the **2D_SIZE** parameter.
- The amount of words to be skipped between the rows is set by the **SKIP_SIZE** parameter.

Note For more information on Segmented mode, see the Segmented mode section in this document.

Selecting **Use Port memory pointer** adds a read-only address bus to the design. This bus indicates the memory address currently being accessed. The bus is named as follows: ***Ptr_[portname]_to_[destination]***, where **portname** is the name of the current Port and **destination** is the name of the subdesign that is connected to this bus. This bus may be connected to user's subdesign (entity/module) or to the software via PCI/E. It may be used to check the Port status (current location) in the real time.

Clicking **Pointer destination** opens the **Units** dialog box (see Figure 9). Here you can set the subdesigns (Modules / Entities) that will be connected to the pointer bus. In addition, you must define the clock domain for bus synchronization.



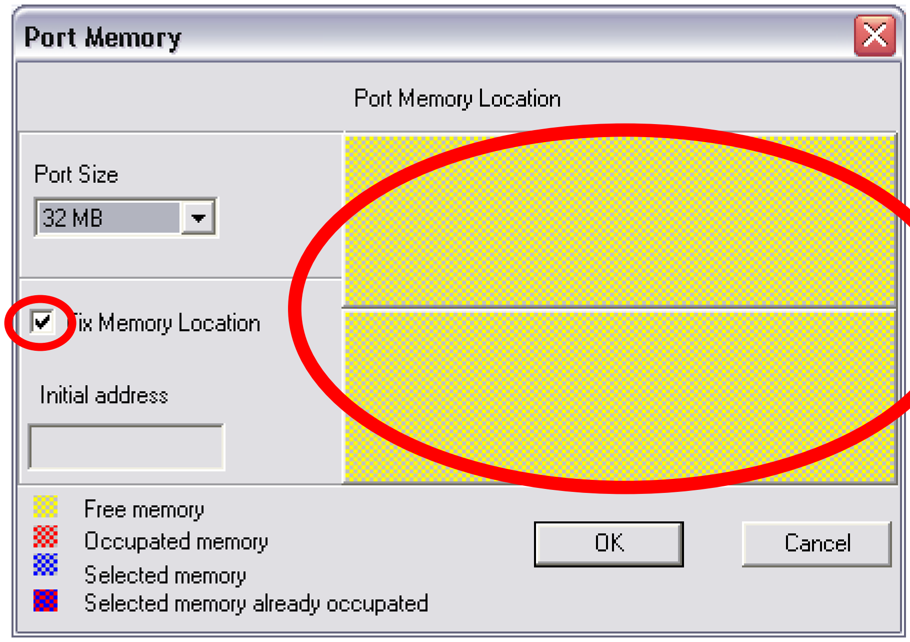
Status	Name	Clock
☐	PCI	lclk
✓	timer_unit	clk

Select destination clock domain: clk

Figure 9 – Port Memory Pointer Destination Dialog

Choosing **Limit Port memory size** enables to logically set the memory size to be accessed by the port. The memory size must be a power of 2, starting from 512 Bytes and up to half the physical memory size. Choosing this option will make the Port occupy only a part of the memory, while the rest of the memory space remains available for other usages (or other Ports).

A size-limited Port may be restricted to various locations in the physical memory space. You may choose whether to fix this location or allow it to change dynamically. If the memory location is fixed and larger than MEMORY SIZE / 16, you may choose the location visually, as shown in Figure 10. In this example, the Port is limited to half the memory size. Clicking on the upper half will force the Port to start from address 0. Clicking on the bottom half will force the Port to start from the 32nd MB.



Port Size: 32 MB

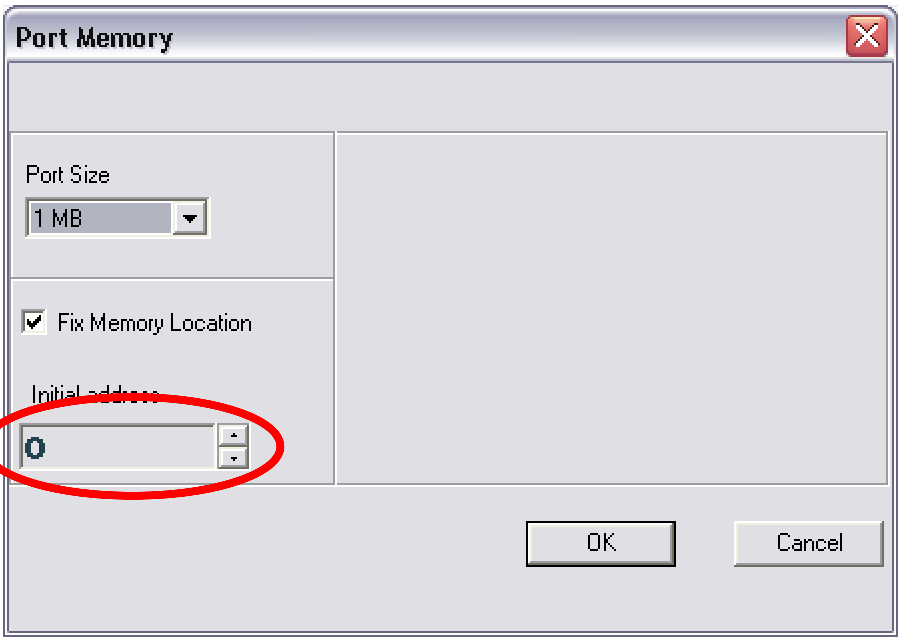
[✓] Fix Memory Location

Initial address

🟨	Free memory
🟥	Occupated memory
🟦	Selected memory
🟪	Selected memory already occupated

Figure 10 – Setting Port Memory Location Visually

If the **Port size** is less than MEMORY SIZE / 16, the graphical interface may not be used as the rows would be too small. Therefore, the manual Initial address setting must be used, as shown in the figure below.



Port Size: 1 MB

[✓] Fix Memory Location

Initial address: 0

Figure 11 – Setting Port Memory Location Manually

By unchecking the **Fix Memory Location** feature, as shown in the Figure 12, the port's memory location will be dynamic. In this case, the memory part accessed by the Port is set in real-time by setting the unused MSbits of Port address bus.

For example, if the memory data width is 64bits (memory word is 8 bytes wide) and the memory size is 32Mbytes (4 MegaWords), then the memory address bus is LOG2(4MegaWords) = 22bits. If a certain Port is limited to 8Mbytes (1 MegaWord) of the memory address space, then the address space for this Port is LOG2(1MegaWord) = 20 bits. If the memory location is not fixed for this port, its address bus is 22 bits wide, where address bits [19..0] are used for Port addressing and the remaining address bits [21..20] are used to dynamically set the Port location, as follows:

00 – starting at 0 (from 0 to 8MB-1);

01 – starting at 8th Mbyte (from 8MB to 16MB-1);

10 – starting at 16th Mbyte (from 16MB to 24MB-1);

11 – starting at 24th Mbyte (from 24MB to 32MB-1);

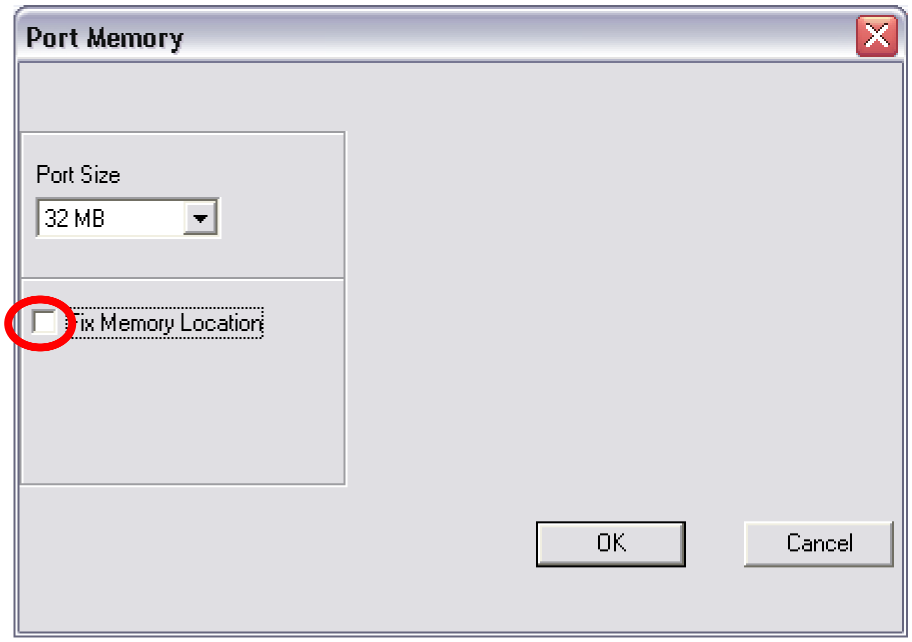
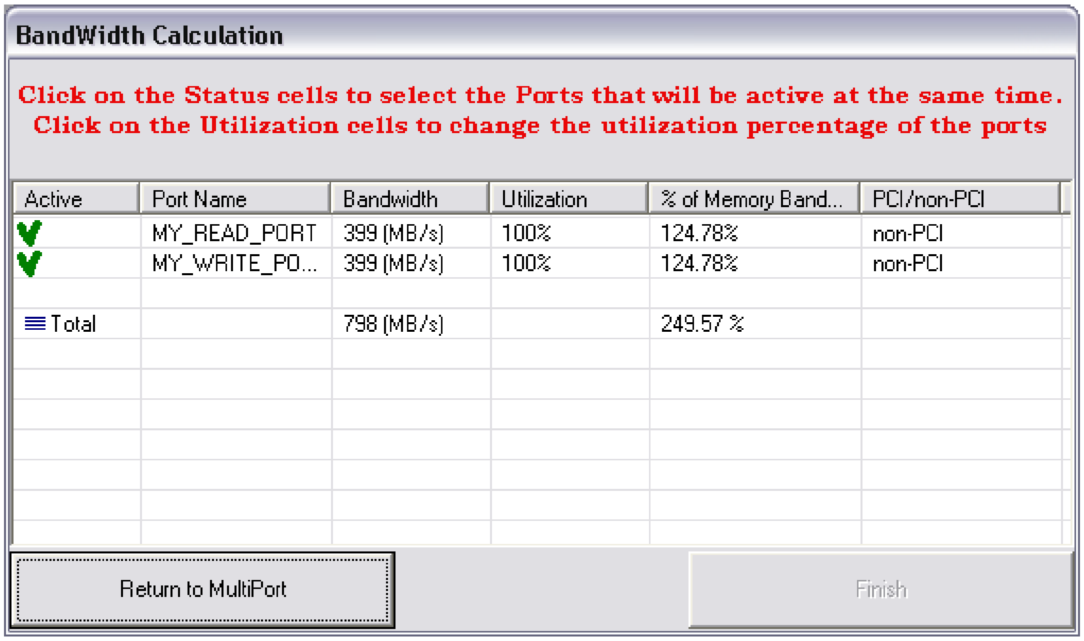


Figure 12 – Port Memory Setting with Non-Fixed Location

Bandwidth Calculation

After you have completed setting-up all the Ports, if the requested bandwidth is greater than the maximal memory bandwidth, the **Bandwidth Calculation** dialog box (Figure 13) will appear, displaying the resulting bandwidth for each Port.



Click on the Status cells to select the Ports that will be active at the same time.
Click on the Utilization cells to change the utilization percentage of the ports

Active	Port Name	Bandwidth	Utilization	% of Memory Band...	PCI/non-PCI
✓	MY_READ_PORT	399 (MB/s)	100%	124.78%	non-PCI
✓	MY_WRITE_PO...	399 (MB/s)	100%	124.78%	non-PCI
≡ Total		798 (MB/s)		249.57 %	

Figure 13 – Bandwidth Calculation Dialog Box

In the **Bandwidth Calculation** dialog box, you can try different active DMA combinations and work durations to achieve optimal performance. This is an information window only and its settings will not affect the HDL code generation. However, you will not be allowed to perform generation if the defined bandwidth exceeds the maximal memory bandwidth.

To configure the ports to fit the maximal Bandwidth limitation, you may change either the **Utilization** allocated to each port by clicking in the **Utilization** column the specific port's cell, or the active ports working concurrently by clicking in the **Active** column the specific port's cell.

Bandwidth is calculated using the following formula:

Bandwidth = 75% × [(memory clock frequency) × (memory width)]

For example: 32bit-bus memory runs on 150MHz:

Bandwidth = $0.75\times(150\times10^6\times32)$ = 3.6Gb/s = 450MB/s

If the memory is of a DDR type, the final throughput is doubled:

Bandwidth = 3.6Gb/s×2 = 7.2Gb/s = 900MB/s

The sum of all active ports bandwidths should be less than the maximum memory bandwidth.

On new Stratix III/IV PROC boards, there is an Automatic Memory Bandwidth Controller. With this feature, the PCI/e ports are able to reduce automatically their data rate in order to keep the memory bandwidth at 100%.

Clicking **Finish** adds the ***PROCMultiPort*** to the user design with all the required components (memory status registers, memories and register groups).

Note A MultiPort with all its elements can be cut and pasted or copied and pasted to another IC.

Memory Bandwidth Controller

The Memory Bandwidth Controller is a very important feature that allows you to utilize fully the total memory bandwidth on the PROC boards.

The total bandwidth required by the user is the bandwidth used by each Port multiplied by the time fraction during which this Port is active. For example, if the user has defined two ports, each using 100% of the memory bandwidth, and each port is active during 50% of time, then the total memory bandwidth used will be 100%.

With the Automatic Memory Bandwidth Control feature on, the PCI/E ports will automatically hold and resume the DMA operation in order to keep the total required memory bandwidth no more than 100%.

In ***PROCWizard* Debug mode**, you can access **MultiPort** subdesign components (registers, memories and register groups). The Debug Mode is used to debug designs in the real-time. Data can be written / read on-the-fly to / from the registers and memories that appear in the design. To enter the Debug mode, click the button in ***PROCWizard's*** toolbar, as shown in the figure below:

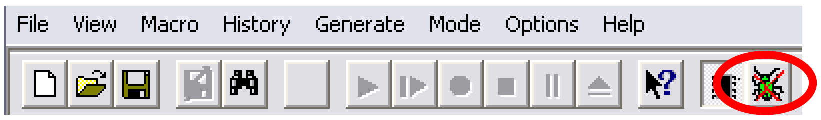
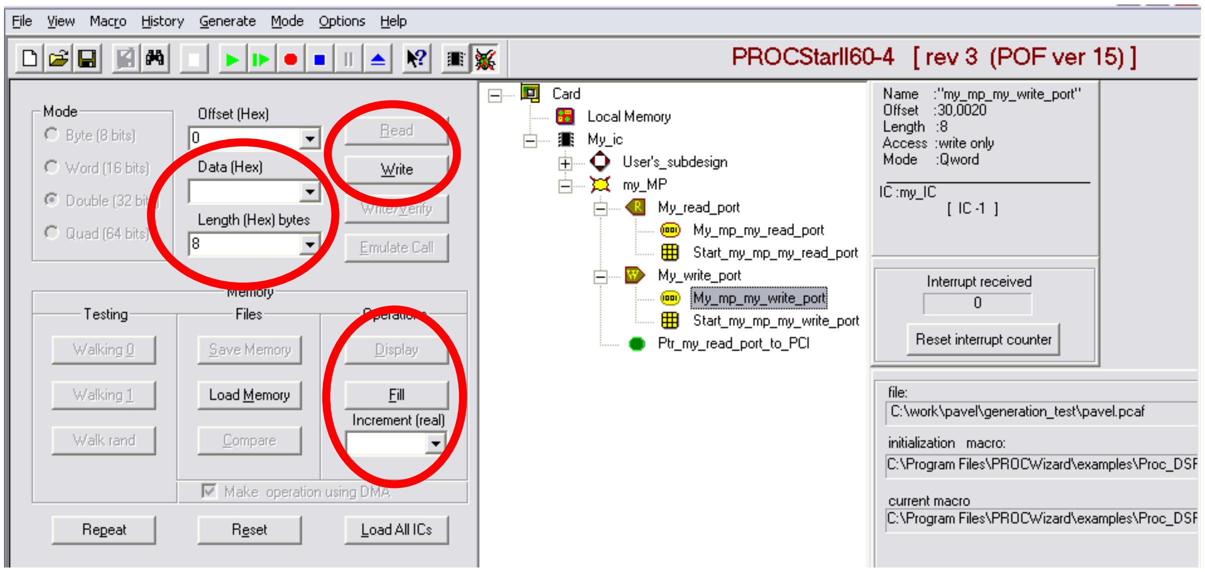


Figure 14 – Debug Mode Icon

Once in Debug Mode, you will see the design you are building represented by an item tree, as shown in the figure below. To access registers/memories, use the **Read** and **Write** buttons and the **Data(Hex)** field.



Offset (Hex): 0
Length (Hex) bytes: 8

Name :"my_mp_my_write_port"
Offset :30,0020
Length :8
Access :write only
Mode :Qword

IC :my_IC
[IC -1]

Interrupt received: 0

file:
C:\work\pavel\generation_test\pavel.pcaf
initialization macro:
C:\Program Files\PROCWizard\examples\Proc_DSF
current macro
C:\Program Files\PROCWizard\examples\Proc_DSF

Figure 15 – PROCWizard Debug Mode Window with PROCMultiPort IP

The following **MultiPort** subdesign items (registers and memories) enable to debug the memory:

- Register **Start_[*multiportname]_[portname]*** (see Figure 15) is used to reset the Port. Writing the required starting address to this register asserts the Start signal high for that Port and at the same time places the address on the Port address bus. This resets the Port and returns it to the specified address.
- Memory [***multiportname]_[portname]*** is the Port FIFO. Data can be read / written to this memory, thus accessing the on-board memory.

Note PROCWizard always uses hexadecimal values when reading / writing registers / memories.

In addition, it is possible to write large portions of data to Ports to test long transfers. To do this, set the transfer length in the **Length** field. Define the starting value and increment value by setting the **Data** and **Increment** fields. Then click **Fill**. This will transfer data to a WRITE Port. Clicking **Display** will display data from a READ Port.

To learn more about debugging designs using PROCWizard, please refer to chapter 6 in **PROCWizard User Manual**.

This chapter describes the parameters and signals of ***PROCMultiPort SDRAM Controller***, as they appear in the top-level design generated by GiDEL ***PROCWizard.***

Parameters

The parameters are divided in two independent groups: **Memory Parameters** and **Port Parameters**. These parameters must be provided to the controller's core at instantiation.

Memory Parameters

The following parameters specify the working mode of the SDRAM memory.

Table 1: Memory parameters

Parameter	Meaning
CLK_FREQ	Memory operating frequency.
MEM_ADDR_WIDTH	Memory address bus width. The number of accessible addresses is $2^{(MEM\_ADDR\_WIDTH)}$.
MEM_WIDTH[(1)]	Memory data bus width. The memory size in bytes is $2^{(MEM\_ADDR\_WIDTH)} \times (2\times MEM\_WIDTH) / 8$
NUMBER_OF_PORTS	Number of Ports that access the memory.

(1) The provided calculations refer to DDR memory where the memory bus width is doubled. Exclude **"2×"** from the formula for SDR memory.

Port Parameters

Each Port has its own **Sequential Name** and its own set of **Port Parameters**.

- **Sequential Names** are the capital letters A to P. Port 0 is associated with the letter A. Port 1 is associated with the letter B, etc.
- **Port Parameters** are independent for each Port. The following table describes the Port Parameters

Table 2: Port parameters

Parameter name (X= port sequential name)	**Usage**	**Notes**
PORT_[X]_WIDTH	Port data width. Should be set to **0** for unused Ports*(3)*	Should be specified for every Port
PORT_[X]_ACCESS	Access method: **RANDOM** or **SEQUENTIAL**	Should be specified for each used Port.
PORT_[X]_DIRECTION	Sequential Port direction: **READ** or **WRITE**	Should be specified for each used sequential Port. Ignored for **RANDOM** Ports.
PORT_[X]_ADDR_WIDTH	Limits the size of the Port. For example: ADDR_WIDTH = 21 means that the address bus of the current Port is 21bit wide *(4)*	If not specified, set to **MEM_ADDR_WIDTH** by default.
[X]_DMA	Sequential Port DMA mode: **DISABLE** or **ENABLE**	If not specified, set to **DISABLE** by default. Ignored for **RANDOM** Ports
[X]_1D_SIZE	Segmented mode row length. Must satisfy the following constrain: **1D_SIZE= N•(K1•K2•MEM_WIDTH)/ WIDTH** Where: **N** ≥4. **K1** = 2 for DDR memory, else 1. **K2** = 2 for Half rate mode, else 1where N is not less than 4	Should be specified for each **SEQUENTIAL Segmented** Port. Ignored for **RANDOM** Ports.

Parameter name (X= port sequential name)	**Usage**	**Notes**
[X]_2D_SIZE	Number of rows in Segmented mode.	Should be specified for each **SEQUENTIAL Segmented** Port. Ignored for **RANDOM** Ports.
[X]_SKIP_SIZE	The gap between two adjacent rows in Segmented mode. Must satisfy the following constrain: **SKIP_SIZE= M•(K1•K2•MEM_WIDTH)/ WIDTH** Where: **M** = positive integer. **K1** = 2 for DDR memory, else 1. **K2** = 2 for Half rate mode, else 1	Should be specified for each **SEQUENTIAL Segmented** Port. Ignored for **RANDOM** Ports.
[X]_DMA_DYNAMIC_SIZE	Segmented mode parameters (1D_SIZE, 2D_SIZE and SKIP_SIZE) to be dynamically changed: **DISABLE** or **ENABLE**; If not specified, set by default to **DISABLE**. Ignored for RANDOM Ports.	

Notes

(1) This is the only parameter that should be specified for unused Ports as well. Unused Ports have 0 width. Other parameters are ignored if this parameter set to zero. First <**NUMBER_OF_PORTS**> Ports should have this parameter greater than zero. **RANDOM** Ports can be <(**2× MEM_WIDTH**)> wide only. **SEQUENTIAL**Port width must satisfy the following equation: **WIDTH** = (**2× MEM_WIDTH**) **/ (2^N)**, where **N** is a positive integer.

(2) For example, in case of DDR memory, if **MEM_WIDTH =32** and **ADDR_WIDTH=21** then the current Port size will be 2^{21} * 64 / 8 = 16MB. If the physical memory size is 32MB then the Port is limited to half the memory size. Define which part of the memory the Port will use by setting the **addr_base** signal of the limited Port.

Signals

This section describes the signals of each interface.

Memory Signals

The following signals are used to control the memory.

Table 3: Memory signals

Signal	Dir	Functionality
mem_clk	I	Clock signal that memory is synchronized with
shifted_mem_clk	I	Clock signal shifted by -90 deg relatively to mem_clk
ref_clk	I	Clock signal for DQS logic. Same freq. as mem_clk
cs	O	Memory chip select signal
ras	O	Memory row select signal
cas	O	Memory column select signal
ce	O	Memory clock enable signal
we	O	Memory write enable signal
dqm[]	O	Memory byte enable signals (byte mask)
ba[]	O	Memory bank selection signals
addr[]	O	Memory address bus
data[]	I/O	Memory data bus
cs_bus	O	Memory chip select bus for SODIMM memory modules

All these signals are connected directly to the memory. Signal **mem_clk** must use the same clock used by the memory.

Port Signals

The following table describes the signals of one Port. Each Port has the same set of signals. The settings are independent for all the Ports.

Table 4: Port signals

Signal	Functionality	Dir	Applicable for…
clrn	Global reset signal. Active low.	I	Global MultiPort signal
clk	Port clock domain. Controls and I/Os are synchronized with clk	I	Applicable in all modes
port_addr	Access address for **Random** Ports Starting address for **Sequential** Ports	I	Applicable in all modes
addr_base	Dynamically chooses the part of physical memory to connect the Port to.	I	For **limited** Ports*(1)*
port_start	**Asserting low:** forces the Port to go to a new address. The new address is read from **port_addr** register*(2)* **Asserting high:** erases the internal FIFO.	I	**Sequential** Ports only
port_[X]_data_in	Data to be written into memory via Port **X**, where **X** is a, b, c etc.	I	**Random** Ports **Sequential WRITE** Ports
port_[X]_data_out	Data read from memory via Port **X**, where **X** is a, b, c etc.	O	**Random** Ports **Sequential READ** Ports
port_write	High for **write** access, low for **read** access	I	**Random** Ports only
port_select	When high, performs access to the memory / Port FIFO on the current clock In **Sequential WRITE** mode, **port_select** must remain low until **almost_full** asserts low. In **Sequential READ** mode, **port_select** must remain low until **almost_empty** asserts low.	I	Applicable in all modes
port_ready	In **Random** Port, goes high when access to memory ends. In **Segmented** mode, goes high after the whole segment processed.	O	**Random** Ports **Segmented Sequential** Ports
port_flush	If sampled high, pushes the last value from the internal FIFO to the memory. May be asserted on termination of write bursts to make sure that all the data reaches the memory *(3)*	I	**Sequential WRITE** Ports only
empty	Goes high when the Port FIFO is empty.	O	**Sequential** Ports only
almost_empty	Goes high when the Port FIFO is less than 1/8 full. For READ Ports, indicates whether data is available for reading from FIFO.	O	**Sequential** Ports only
almost_full	Goes high when the Port FIFO is more than 7/8 full. For WRITE Port, indicates that pause should be inserted in data transfer so that the Port will have time to drain the FIFO.	O	**Sequential** Ports only

Signal	Functionality	Dir	Applicable for…
port_internal_addr	The current Port address. Incremented when the controller accesses the memory serving the Port (4)	O	**Sequential** Ports only
ptr_enable	Allows synchronizing internal address with user clock (goes high when **port_internal_addr** is available for reading)	O	**Sequential** Ports only
port_1d_size	Segmented mode row length. Must satisfy the following constrain: **port_1d_size = N × (K1 × K2 × MEM_WIDTH) / WIDTH** Where **N** ≥4, **K1** = 2 for DDR memory, else 1; **K2** = 2 for Half rate mode, else 1.	I	**Segmented Sequential** Ports (DMA Dynamic size only)
port_2d_size	Number of rows in Segmented mode.	I	**Segmented Sequential** Ports (DMA Dynamic size only)
port_skip_size	**port_skip_size = M × (K1 × K2 × MEM_WIDTH) / WIDTH** Where **M** is a positive integer, **K1** = 2 for DDR memory, else 1; **K2** = 2 for Half rate mode, else 1.	I	**Segmented Sequential** Ports (DMA Dynamic size only)
port_error	Asserts high when the contents are corrupt due to writing to a full FIFO (for a WRITE port) or due to reading from an empty FIFO (for READ port).	O	**Sequential** port only.

Table 5: Port signals

Notes:

(1) Ports for which **PORT_[X]_ADDR_WIDTH** < **MEM_ADDR_WIDTH** (See Limiting Port Size paragraph)

(2) Should be asserted high till sampled at least once on rising edges of both “mem_clk” and “clk”. **Do not begin new transaction until the previous one has completed.**

(3) See Using the port_flush Signal paragraph in this section.

(4) This address is not necessarily incremented each time the **Port_select** signal is asserted high. For example, if the memory width is 128 bits and the port width is 32 bits, only after 128/32=4 writes/reads the address pointer will be incremented.

Limiting Port Size

To limit the size of a certain Port, you must define the size of that Port using the **PORT_[X]_ADDR_WIDTH** parameter. For example, suppose Port 0 (port A) is 8bits wide. Then, to limit it to 1MB, set **PORT_A_ADDR_WIDTH** to 20 (for more information, see the Port Parameters section in this document).

If **PORT_[X]_ADDR_WIDTH** is set to be less than **MEM_ADDR_WIDTH**, then the port may access only a part if the memory space. The part of the memory that will be used by that Port is defined by the **addr_base** bus.

The size of memory accessed by Port **X** can be calculated as follows:

PORT_[X]_SIZE = $(2^{\textbf{ADDR}\_\text{WIDTH}})$*(**PORT_[X]_WIDTH** / 8) (in bytes).

For example:

PORT_A_WIDTH = 64	(port 0 is 64bits wide)
PORT_C_WIDTH = 8	(port 2 is 8bits wide)

To limit Port 0 to 16MB, set **PORT_A_ADDR_WIDTH = 21**	$(2^{21}$ * 64 / 8 = 16Mbytes)
To limit Port 2 to 1MB, set **PORT_C_ADDR_WIDTH = 20**	$(2^{20}$ * 8 / 8 = 1Mbyte)

Connect **addr_base** inputs to logic (or constants) to define which part of the physical memory will be used by that Port.

For example, if physical memory size is 32MB, then for the example above:

```
addr_base[0] [] = 0;    -- first half (16MB) of memory
addr_base[0] [] = 1;    -- second half (16MB) of memory

addr_base[2] [] = 0;    -- first 1/32 (1MB) of memory
addr_base[2] [] = 1;    -- second 1/32 (1MB) of memory
addr_base[2] [] = 2;    -- third   1/32 (1MB) of memory
addr_base[2] [] = 31;    -- last 1/32 (1MB) of memory
```

If Port location is fixed, **addr_base** will be automatically connected to a constant according to the specified location. If the location is not fixed, **addr_base** will be connected to the Most Significant Bits of the Port's Address bus. In this case, Port location may be selected by setting the MSBits of that Port Address bus to the desired value.

Using the **port_flush** Signal

Sequential WRITE Ports have an additional buffer placed before Port's internal FIFO. This buffer is used when the Port data width is less than the memory data width. In this case, several write accesses to the Port are needed to complete a whole memory word. For example, if memory width is 32bits and Port width is 8bits, then four writes to the Port are needed to write one word to memory.

The buffer before the internal FIFO accumulates these accesses to form a memory word from several (smaller) Port words.

To push a complete memory word out of the buffer, one must start writing the next word. In the above example, four 8-bit words written to the Port will form one 32-bit memory word, but only writing the fifth 8-bit word will push the previous memory word into Port FIFO and further to the memory.

Asserting **port_flush** high pushes the word from the buffer into the FIFO, even if the memory word in the buffer is incomplete. In such case, the incomplete memory word is padded with zeros and pushed to the FIFO.

port_flush signal is most useful for write burst transfers. If write burst ends before the last memory word is completed (for example, 5 eight-bit words were written in the above case), you can assert a high **port_flush** signal to pad the data with zeros and push the last word to memory.

Nevertheless, it is preferable that you write an additional memory word manually to push the last data from the buffer to the memory.

Each Port of ***PROCMultiPort*** may be configured to work in one of two modes: **Random** or **Sequential**. In **Sequential** mode, the Port may be configured to work in a special way called **Segmented** mode.

Random mode

The **Random** mode enables to perform random read and write operations through the Port. Each time the Port accesses the memory using a specific address – bursts cannot be implemented with a Random Port. Random Ports have the highest priority when accessing the memory.

To access data using a **Random** Port, simply load the data to the Port Address Bus (**port_addr**), select direction using **port_write** signal (high for writing, low for reading) and write/read data to/from the Port Data Bus (**port_X_data_in / port_X_data_out**), where X stands for the Port name (A, B, C etc.). Assert **port_select** high to initiate access to memory. When the Port completes the memory access, it will raise the **port_ready** signal.

Sequential Mode

Sequential mode enables the to perform bursts to and from the on-board memory. **Sequential** Ports may have one direction only: they must be defined either as READ or WRITE. The following figure describes how the Port accesses memory.

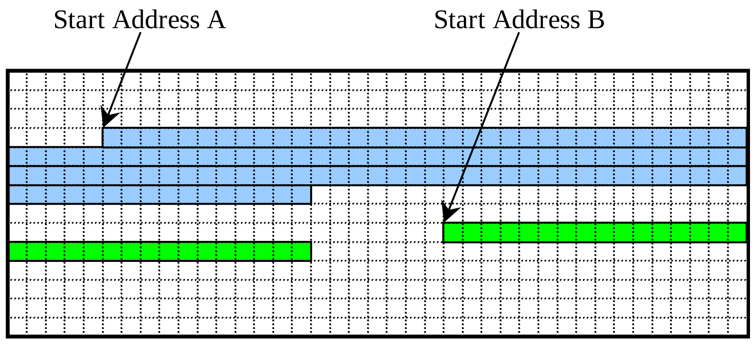


Figure 16: Sequential Port accesses memory

All Sequential Ports have a FIFO interface. The data is not read/written directly from/to memory but goes through an internal FIFO implemented inside the controller.

Sequential Ports have **almost_empty**, **almost_full** and **empty** flags driven by the internal FIFOs. These flags enable to control the status of the Port in order to maintain proper FIFO operation.

A **Sequential** Port configured behaves according to the following scheme:

```
repeat always
{
    if  port_start == VCC  then
        port_internal_addr = port_addr;                  // "start" sampled high
    else
    {
        if  FIFO is more than 7/8 full   then   almost_full = VCC;
        if  FIFO is less than 1/8 full   then   almost_empty = VCC;
        if  FIFO is empty                then   empty = VCC;

        if  ACCESS == READ   &  FIFO has some empty space   then
        {
            read Port_internal_addr;                     // read from memory to FIFO
            increment Port_internal_addr;
        }

        if  ACCESS == WRITE   &   FIFO has some data   then
        {
            write Port_internal_addr;                    // write from FIFO to memory
            increment Port_internal_addr;
        }

        if  port_select == VCC   then
        {
            if  ACCESS == READ    then   read data_from_FIFO;    // output data to user at each clock
            if  ACCESS == WRITE   then   write data_to_FIFO;     // read data from user at each clock
        }

    }
}
```

Note

The **almost_empty** signal is of highest importance when a **Sequential** Port is in the **READ** mode. When this signal goes low, this means that the FIFO is more than 1/8 full, i.e. the controller has filled the FIFO with some data from memory. When **almost_empty** goes low, the Port may be read. In other words, the **port_select** signal of the read Port should remain low until the **almost_empty** signal goes low, meaning that FIFO is not empty anymore. This ensures proper FIFO operation.

Reading from a Port

To read from a port, perform the following:

1. Load **port_addr** with starting address.
2. Assert the **port_start** signal high. This will reset the Port.
3. Assert the **port_start** signal low. This will make the Port start reading data from the memory to internal FIFO. Wait for some data to be read to internal FIFO. The Port must not be read during this time. Generally, the delay needed may be calculated using the following formula:

   delay = 16+ (FIFO depth * 1/8) [memory clocks]

   Where **FIFO depth** is dependent on board type and on the memory bank used as detailed in the following table:

Table 6: FIFO Depth

Board	Memory Bank	FIFO Depth	Width
PROCe IV **PROCe III**	A, B, C	256 words	256 bits
PROCStar III	A, B	256 words	256 bits
PROCStar III	C	256 words	128 bits
PROCStar II	A, B	128 words	64 bits
PROCStar II	C	128 words	128 bits
PROCSpark II	A	128 words	64 bits
PROCe	A, B	128 words	64 bits

4. Assert the **port_select** signal high to start reading the Port.

Instead of calculating this delay, one can poll the **almost_empty** signal. When the Port is 1/8 full, **almost_empty** asserts low indicating that the **port_select** signal can be asserted high. The following diagram illustrates the read timing diagram:

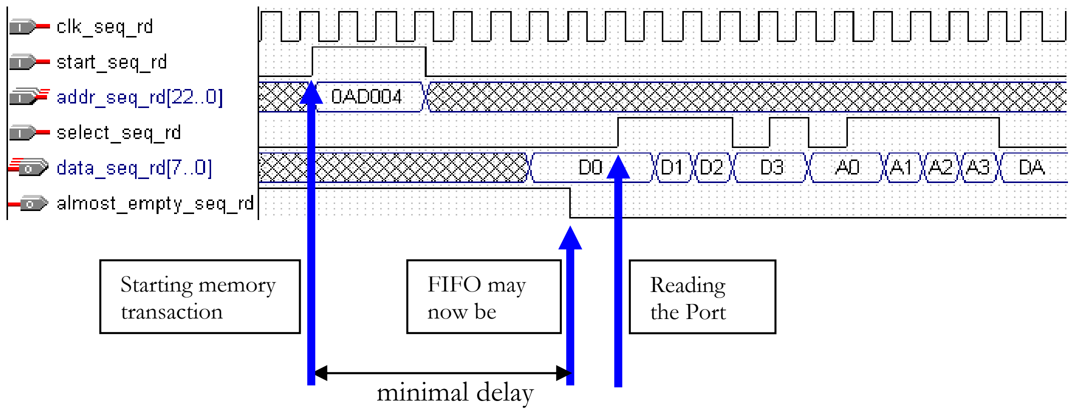


Figure 17: Reading From a Sequential Port

Writing to a Port

To write to a Port, perform the following steps:

1. Assert the **port_start** signal high. This will reset the Port.
2. Assert the **port_start** signal low. This will set the starting address of the Port.
3. Wait until the **almost_full signal** asserts low. The Port is now ready to receive data from user's logic. The Port will now also start writing data from the internal FIFO to the memory.
4. Assert the **port_select** signal high to start writing to the Port. The Port will immediately start reading data from user's logic to internal FIFO. The Port will expect new data at each Port clock.

If the memory throughput is not fast enough, the internal Port FIFO will eventually become full. In this case, one can poll the **almost_full** signal. When the Port is 7/8 full, the **almost_full** will assert high indicating that a pause must be asserted in the data sequence and that the **port_select** signal must be asserted low. The following d illustrated write timing:.

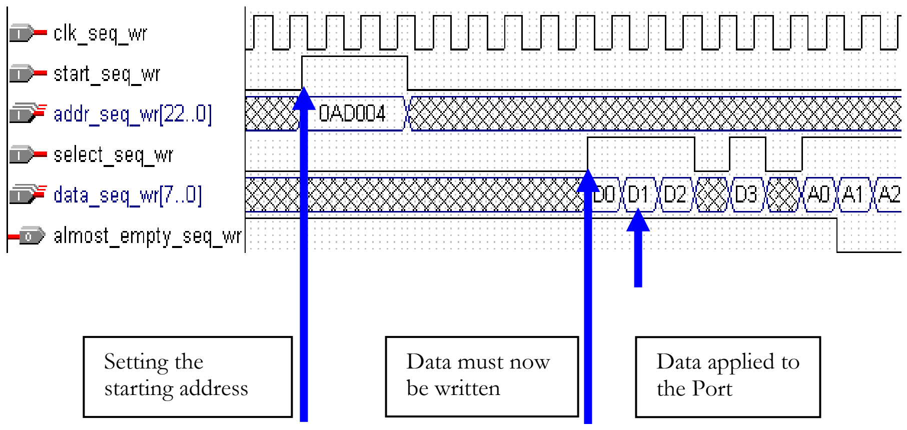


Figure 18: Writing To a Sequential Port

Synchronizing data transfer between PROC boards

Assume that we need to transfer some data from one PROC board to another. In order to transfer the data properly, either both the boards must work at exactly the same frequency (e.g., be driven from a single external clock source), or a FIFO must be introduced between the boards. Using ***PROCMultiPort*** eliminates the need to introduce an additional FIFO.

In order to transfer data, set up a **READ** Port on the source board and a **WRITE** Port on the target board. Connect the target Port clock to the source Port clock. This will synchronize the data transfer between the Ports, still allowing the on-board memories to work at different frequencies.

Now connect the **almost empty** signal of the source Port and the **almost full** signal of the target Port to the **select** signal of both the Ports, as shown in the picture below. This way the data transfer will be delayed automatically if the target Port FIFO has become full (the source board is faster) or if the source Port FIFO has become empty (the target board is faster).

In example below, data transfer will start upon asserting the **Go** signal high and stop when all the data will be transferred (or when the **Go** signal will go low).

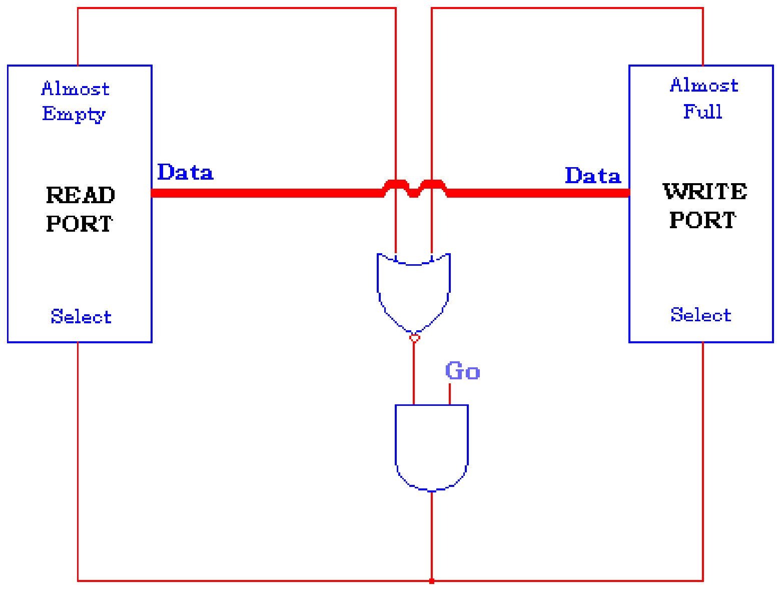


Figure 19: Automatic Data Transfer Synchronization

Segmented mode

This special mode enables to perform 2D data transfers. In this mode, the Port accesses several words (a row) sequentially and then skips several words. After the Port finishes processing all the rows, it asserts the **ready** signal high.

- The row size is set by the **1D_SIZE** parameter.
- The number of rows to be accessed is set by the **2D_SIZE** parameter.
- The amount of words to be skipped is set by the **SKIP_SIZE** parameter.

To start accessing memory in the **Segmented** mode, apply the segment starting address to the **port_addr** register and assert **port_start** high. On the following accesses to the Port (when **port_select** will go high), the controller will behave according to following scheme:

```
repeat always
{
   if  port_start == VCC  then
       port_internal_addr = port_addr;                // "start" sampled high
   else
   {
       if  FIFO is more than 7/8 full   then  almost_full = VCC;
       if  FIFO is less than 1/8 full   then  almost_empty = VCC;
       if  FIFO is empty                then  empty = VCC;

       for (i = 0; i < 2D_SIZE; i++)                  // read window line by line
       {
          for (j = 0; j < 1D_SIZE; j++)               // read a single line from memory
          {
             if  ACCESS == READ   &  FIFO has some empty space  then
             {
                 read port_internal_addr;             // read from memory to FIFO
                 increment port_internal_addr;
             }

             if  ACCESS == WRITE   &  FIFO has some data  then
             {
                 write port_internal_addr;            // write from FIFO to memory
                 increment port_internal_addr;
             }

             if  select == VCC  then
             {
                 if  ACCESS == READ   then  read data_from_FIFO;   // output data to user at each clock
                 if  ACCESS == WRITE  then  write data_to_FIFO;    // read data from user at each clock
             }
          }
          port_internal_addr = port_internal_addr  + SKIP_SIZE;   // Finished reading line, skip the rest of it
       }   // Finished reading window

       port_ready = VCC;
   }
}
```

When the window has been entirely read/written, the **ready** signal goes high. Further accesses to the Port (asserting **select** high) will result in Port exceeding the defined window size. Therefore, you must stop accessing the Port when **ready** signal goes high.

If **DMA_DYNAMIC_SIZE** is selected, set the values for row length (**1d_size**), number of row (**2d_size**) and gap size between two adjacent rows (**skip_size**) before asserting **port_start** high.

Window Processing

Assume that we need to process an image on a N x M window. The following figure represents the memory.

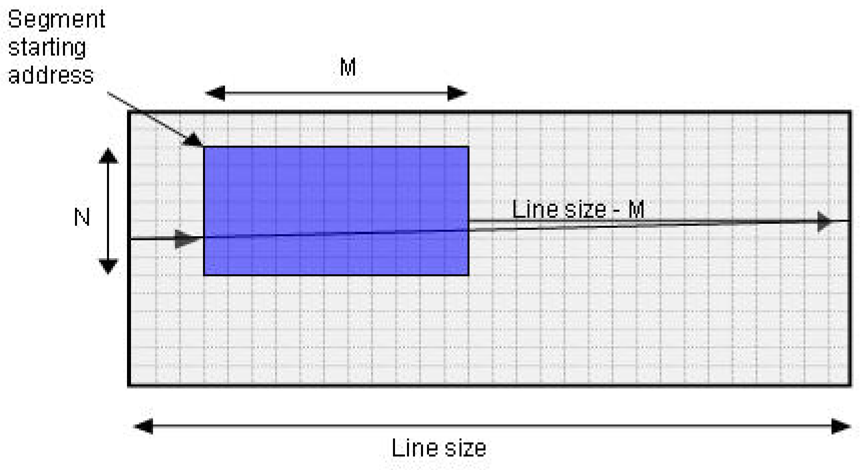


Figure 20: Window Processing in Segmented Mode

In order to access only the colored segment of memory, set up the Port with the following parameters:

- **1D_SIZE** = M
- **2D_SIZE** = N
- **SKIP_SIZE** = Line size - M

The starting address of the segment must be applied to the **port_addr** register of the Port when the **port_start** signal of the Port is sampled high.

Going Backwards

In the Segmented Mode, you can store data in a different order than it arrives. For example, image lines can be stored in backward order (last line received first, first line received last).

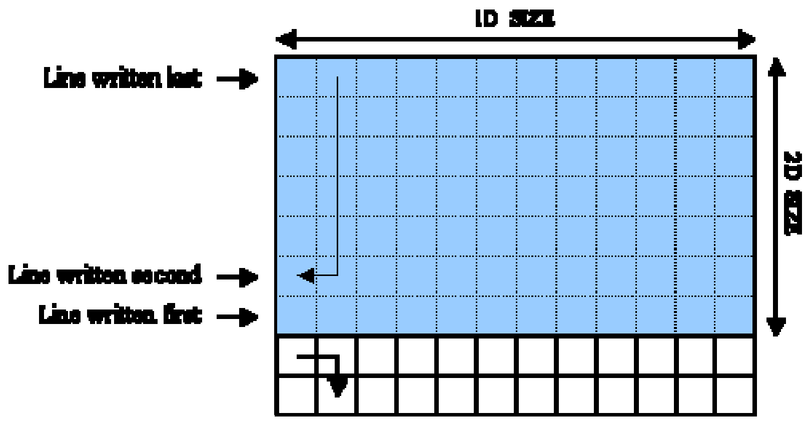


Figure 21: Going Backwards in Segmented Mode

We start with address pointer at the beginning of the last line. After the line has been written, the address pointer will point beyond the frame. Skipping minus one line size will place the pointer at beginning of the line just received. Skipping minus 2 line sizes will place the address pointer at the beginning of previous line. The internal memory pointer will now point to the beginning of the next line to arrive.

Skipping the entire memory size would bring us back exactly to the same point (the Port address will wrap-around). In order to reach the correct location for writing the second line, we should skip the entire memory size minus the size of two lines. Thus, **SKIP_SIZE** should be set to (**memory size**) – 2*(**1D_SIZE**), where **memory size** is specified in **PORT_WIDTH** quantities (e.g. if memory size is 32Mbytes and **PORT_WIDTH** is 32 bits = 4 bytes, then **memory size** is 32M / 4 = 8M).

Using this configuration, lines will be written in the order specified in Figure 5.

Two Video Fields

The Segmented mode is a convenient way to handle odd and even fields of video. Assume that you have one Port that writes odd fields and another that writes even fields. Each Port should be configured in the same way but with a different starting address (starting address is applied to the **port_addr** register when the **port_start** signal is sampled high). The first Port would start at the beginning of the first line and the second Port at the beginning of the second line (first odd line).

After writing an entire row, next row should be skipped completely. Therefore, the **1D_SIZE** and **SKIP_SIZE** must both be set to the length of the row.

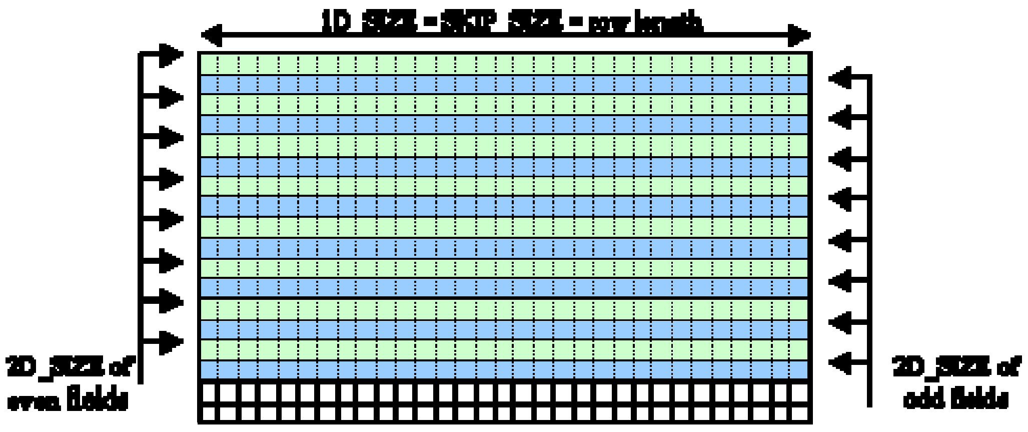


Figure 22: Processing Two Video Fields Using Segmented Mode

A single sequential Port configured as described above can do the same job: first handle even lines (starting address 0) and after the whole field is processed apply a new starting address (which is the beginning of the second row) for the odd lines.

Eliminating External FIFO's

Suppose we wish to perform a kernel operation (18*18) on a medium-size 8-bit image (8K * 8K). For such a process one would normally use 18 line delays of 8 bit * 8K each. For that purport, the typical application would use FIFOs. The required memory would sum up to 18*8*8K bits total. Not all FPGAs may fulfill this memory requirement, so external memory devices should be used (for example 4 external FIFOs 36*8K).

With ***PROCMultiPort*** core, these FIFOs may be replaced with one or two simple memory blocks.

The Concept

The image will be stored in the external memory as it arrives. The kernel operation will be processed on windows (say, 256*256). The windows will overlap to ensure proper operation on the window boundary.

Now the memory that is needed for the kernel operation is 144*256 (the window line width is only 256 bytes and not 8K as before). This memory is obtained by using only 18 LEs in the FPGA (several additional LEs for ***PROCMultiPort*** controller).

Core instantiation

Assume the memory is 32 bit wide.

Port 0 will be used as the write Port receiving the image.

Port 1 will be used as the read Port in **Segmented** mode, as follows.

- **1D_SIZE** = 256 (window width)
- **2D_SIZE** = 256 (window height)
- **SKIP_SIZE** = 8K - 256

Port 2 is used as 32-bit read Port. This Port is used to get added value. Through this Port, the host may read the original image as well. When Port 1 **port_ready** signal goes high, processing of the next window should be started. This technique may be used in many ways to eliminate external memory devices.

This section shows the timing diagrams for each Port type.

Random Port

Write access: all signals are synchronized to the Port clock (clk_random).

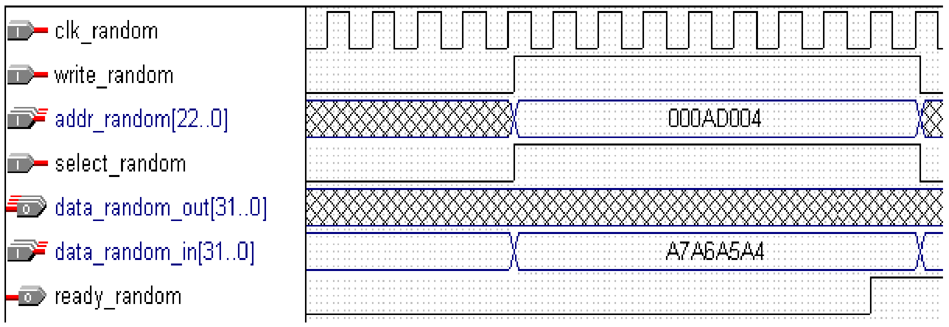


Figure 23: Random Port - Write Access.

Read access: data that was read from memory will be valid until **select_random** goes low.

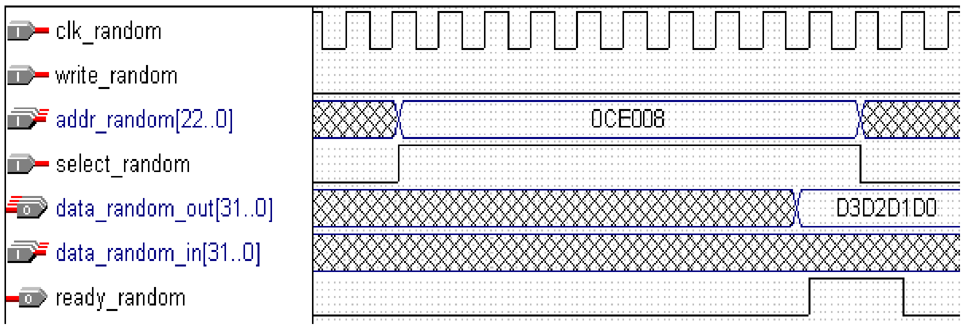


Figure 24: Random Port - read access.

Sequential Port

Sequential WRITE Port. The Port reads data from user’s logic at each Port clock, as long as the **select** signal is high. The data from user’s logic is ignored if the **select** signal is low.

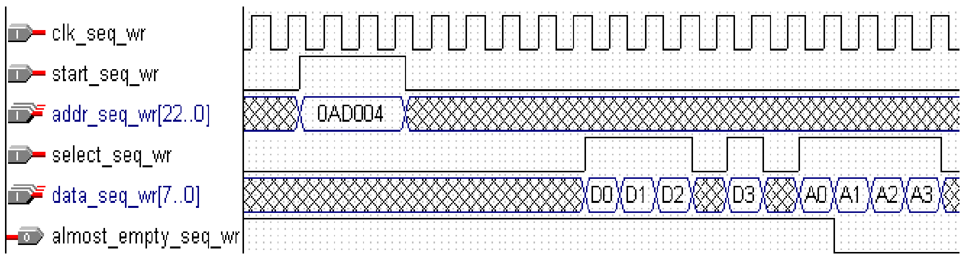


Figure 25: Sequential WRITE Port

Sequential READ Port. The **almost_empty** flag goes low after the controller has read some data from memory into the Port FIFO. The data is read from the Port FIFO as long as the **select** signal is high.

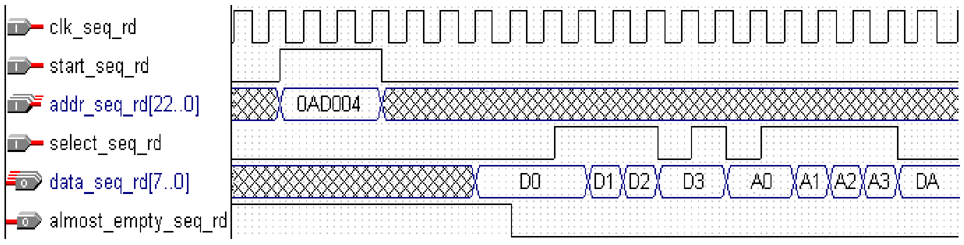


Figure 26: Sequential READ Port

Revision History

Table 7: Document Revision History

Date	**Changes Made**
September 2008	Initial Document
November 2009	• Addition of FIFO depth table • Explanation on Set Record button • Clarification on Bandwidth calculations
January 2010	• Update of Port Parameters and Port Signals tables. • Revision of **Reading from a Port** and **Writing to a Port** • Update of FIFO depth table
June 2010	• Update of port parameters and signals • Update of Segmented Mode section
October 2010	• Deletion of MMR Converter support

Kane Computing Ltd
1 Verdin Street
Northwich
Cheshire
CW9 7BX
Tel: +44(0)1606 351006
Email: sales@kanecomputing.com
Web: www.kanecomputing.co.uk